Minimizing communication in all-pairs shortest paths

Edgar Solomonik
Univ. of California, Berkeley
Department of EECS
solomon@eecs.berkeley.edu

Aydın Buluç
Lawrence Berkeley Nat. Lab.
Computational Research Division
abuluc@lbl.gov

James Demmel
Univ. of California, Berkeley
Department of EECS
demmel@eecs.berkeley.edu

***Abstract*—**

We consider distributed memory algorithms for the all-pairs shortest paths (APSP) problem. Scaling the APSP problem to high concurrencies requires both minimizing inter-processor communication as well as maximizing temporal data locality. The 2.5D APSP algorithm, which is based on the divide-and-conquer paradigm, satisfies both of these requirements: it can utilize any extra available memory to perform asymptotically less communication, and it is rich in semiring matrix multiplications, which have high temporal locality. We start by introducing a block-cyclic 2D (minimal memory) APSP algorithm. With a careful choice of block-size, this algorithm achieves known communication lower-bounds for latency and bandwidth. We extend this 2D block-cyclic algorithm to a 2.5D algorithm, which can use c extra copies of data to reduce the bandwidth cost by a factor of $c^{1/2}$, compared to its 2D counterpart. However, the 2.5D algorithm increases the latency cost by $c^{1/2}$. We provide a tighter lower bound on latency, which dictates that the latency overhead is necessary to reduce bandwidth along the critical path of execution. Our implementation achieves impressive performance and scaling to 24,576 cores of a Cray XE6 supercomputer by utilizing well-tuned intra-node kernels within the distributed memory algorithm.

I. Introduction

The all-pairs shortest paths (APSP) is a fundamental graph problem with many applications in urban planning and simulation [28], datacenter network design [14], metric nearness problem [9], and traffic routing. In fact, APSP and the decrease-only metric nearness problem are equivalent. APSP is also used as a subroutine in other graph algorithms, such as Ullman and Yannakakis's breadth-first search algorithm [41], which is suitable for high diameter graphs.

Given a directed graph $G = (V, E)$ with n vertices $V = \{v_1, v_2, ..., v_n\}$ and m edges $E = \{e_1, e_2, ..., e_m\}$, the distance version of the algorithm computes the length of the shortest path from v_i to v_j for all (v_i, v_j) pairs. The full version also returns the actual paths in the form of a predecessor matrix. Henceforth, we will call the distance-only version by all-pairs shortest distances (APSD) to avoid confusion.

The classical dynamic programming algorithm for APSP is due to Floyd [18] and Warshall [43]. Serial blocked versions of the Floyd-Warshall algorithm have been formulated [32] to increase data locality. The algorithm can also be recast into semiring algebra over vectors and matrices. This vectorized algorithm, attributed to Kleene, is rich in matrix multiplications over the (min,+) semiring. Several theoretical improvements have been made, resulting in subcubic algorithms for the APSD problem. However, in practice, these algorithms are typically not competitive with simpler cubic algorithms.

Variants of the Floyd-Warshall algorithm are most suitable for dense graphs. Johnson's algorithm [25], which is based on repeated application of Dijkstra's single-source shortest path algorithm (SSSP), is theoretically faster than the Floyd-Warshall variants on sufficiently sparse graphs. However, the data dependency structure of this algorithm (and Dijkstra's algorithm in general) make scalable parallelization difficult. SSSP algorithms based on Δ-stepping [31] scale better in practice but their performance is input dependent and scales with $O(m+d\cdot L\cdot \log n)$, where d is the maximum vertex degree and L is the maximum shortest path weight from the source. Consequently, it is likely that a Floyd-Warshall based approach would be competitive even for sparse graphs, as realized on graphical processing units [10].

Given the $\Theta(n^2)$ output of the algorithm, large instances can not be solved on a single node due to memory limitations. Further, a distributed memory approach is favorable over a sequential out-of-core method, because of the high computational complexity of the problem. In this paper, we are concerned with obtaining high performance in a practical implementation by reducing communication cost and increasing data locality through optimized matrix multiplication over semirings.

Communication-avoiding '2.5D' algorithms take advantage of the extra available memory and reduce the bandwidth cost of many algorithms in numerical linear algebra. Generally, 2.5D algorithms can use a factor of c more memory to reduce the bandwidth cost by a factor of $\sqrt{c}$ [37]. The theoretical communication reduction translates to a significant improvement in strong-scalability (scaling processor count with a constant total problem size) on large supercomputers [35].

Our main contributions in this work are:

1) A block-cyclic 2D version of the divide-and-conquer APSP algorithm, which minimizes latency and bandwidth given minimal memory.
2) A 2.5D generalization of the 2D APSP algorithm, which sends a minimal number of messages and words of data given additional available memory.
3) A distributed memory implementation of APSD with

highly tuned intra-node kernels, achieving impressive performance in the highest concurrencies reported in literature (24,576 cores of the Hopper Cray XE6 [1]).

Our algorithms can simultaneously construct the paths themselves, at the expense of doubling the cost, by maintaining a predecessor matrix as classical iterative Floyd-Warshall does. Our divide-and-conquer algorithm essentially performs the same path computation as Floyd-Warshall except with a different schedule. The experiments only report on the distance version to allow easier comparison with prior literature.

The rest of the paper is structured as follows, Section II details previous work on the all-pairs shortest-paths problem. We give a sequential version of the divide-and-conquer APSP algorithm in Section III and provide lower bounds on the bandwidth and latency costs of APSP in Section IV. Section V presents our parallel divide-and-conquer APSP algorithms and Section VI evaluates the scalability and performance of our implementation. We discuss alternative approaches in Section VII and conclude in Section VIII.

II. Previous work

Jenq and Sahni [24] were the first to give a 2D distributed memory algorithm for the APSP problem, based on the original Floyd-Warshall schedule. Since the algorithm does not employ blocking, it has to perform n global synchronizations, resulting in a latency lower bound of $\Omega(n)$. This SUMMA-like algorithm [2], [42] is improved further by Kumar and Singh [27] by using pipelining to avoid global synchronizations. Although they reduced the synchronization costs, both of these algorithms have low data reuse: each processor performs n unblocked rank-1 updates on its local submatrix in sequence. Obtaining high-performance in practice requires increasing temporal locality and is achieved by the blocked divide-and-conquer algorithms we consider in this work.

The main idea behind the divide-and-conquer (DC) algorithm is based on a proof by Aho et al. [4] that shows that costs of semiring matrix multiplication and APSP are asymptotically equivalent in the random access machine (RAM) model of computation. Actual algorithms based on this proof are given by various researchers, with minor differences. Our decision to use the DC algorithm as our starting point is inspired by its demonstrated better cache reuse on CPUs [32], and its impressive performance attained on the many-core graphical processor units [10].

Previously known communication bounds [5], [22], [23] for 'classical' (triple-nested loop) matrix multiplication also apply to our algorithm, because Aho et al.'s proof shows how to get the semiring matrix product for free, given an algorithm to compute the APSP. These lower bounds, however, are not necessarily tight because the converse of their proof (to compute APSP given matrix multiplication) relies on the cost of matrix multiplication being $\Omega(n^2)$, which is true for its RAM complexity but not true for its bandwidth and latency costs. In Section IV, we show that a tighter bound exist for latency, one similar to the latency lower bound of LU decomposition [37].

Seidel [34] showed a way to use fast matrix multiplication algorithms, such as Strassen's algorithm, for the solution of the APSP problem by embedding the $(\min, +)$ semiring into a ring. However, his method only works for undirected and unweighted graphs. We cannot, therefore, utilize the recently discovered communication-optimal Strassen based algorithms [5], [**?**] directly for the general problem.

Habbal et al. [20] gave a parallel APSP algorithm for the Connection Machine CM-2 that proceeds in three stages. Given a decomposition of the graph, the first step constructs SSSP trees from all the 'cutset' (separator) vertices, the second step runs the classical Floyd-Warshall algorithm for each partition independently, and the last step combines these results using 'minisummation' operations that is essentially semiring matrix multiplication. The algorithm's performance depends on the size of separators for balanced partitions. Without good sublinear (say, $O(\sqrt{n})$) separators, the algorithm degenerates into Johnson's algorithm. Almost all graphs, including those from social networks, lack good separators [29]. Note that the number of partitions are independent (and generally much less) from the number of active processors. The algorithm sends $\Theta(n)$ messages and moves $\Theta(n^2)$ words for the 5-point stencil (2-D grid).

A recent distributed algorithm by Holzer and Wattenhofer [21] runs in $O(n)$ communication rounds. Their concept of communication rounds is similar to our latency concept with the distinction that in each communication round, every node can send a message of size at most $O(\log(n))$ to each one of its neighbors. Our cost model clearly differentiates between bandwidth and latency costs without putting a restriction on message sizes. Their algorithm performs breadth-first search from every vertex with carefully chosen starting times. The distributed computing model used in their work, however, is incompatible with ours.

Brickell et al. [9] came up with a linear programming formulation for the APSP problem, by exploiting its equivalence to the decrease-only version of the metric nearness problem (DOMN). Their algorithm runs in $O(n^3)$ time using a Fibonacci heap, and the dual problem can be used to obtain the actual paths. Unfortunately, heaps are inherently sequential data structures that limit parallelism. Since the equivalence between APSP and DOMN goes both ways, our algorithm provides a highly parallel solution to the DOMN problem as well.

A considerable amount of effort has been devoted into precomputing transit nodes that are later used for as shortcuts when calculating shortest paths. The PHAST algorithm [16], which is based on contraction hierarchies [19], exploits this idea to significantly improve SSSP performance on road graphs with non-negative edge weights. The impressive performance achieved on the SSSP problem makes APSP calculation on large road networks feasible by repeatedly applying the PHAST algorithm. These algorithms based on precomputed transit nodes, however, do not dominate the classical algorithms such as Dijkstra and Δ-stepping for general types of inputs. Precomputation yields an unacceptable number of

shortcuts for social networks, making the method inapplicable for networks that do not have good separators. This is analogous to the fill that occurs during sparse Gaussian elimination [33], because both algorithms rely on some sort of vertex elimination.

Due to their similar triple nested structure and data access patterns, APSP, matrix multiplication, and LU decomposition problems are sometimes classified together. The Gaussian elimination paradigm of Chowdhury and Ramachandran [12] provides a cache-oblivious framework for these problems, similar to Toledo's recursive blocked LU factorization [40]. Our APSP work is orthogonal to that of Chowdhury and Ramachandran in the sense we provide distributed memory algorithms that minimize internode communication (both latency and bandwidth), while their method focuses on cache-obliviousness and multithreaded (shared memory) implementation.

A communication-avoiding parallelization of the recursive all-pairs shortest-paths algorithm was given by Tiskin under the BSP theoretical model [39]. Our algorithm is similar, though we pay closer attention to data layout, lower-bound the communication, and study the performance of a high-performance implementation.

Our main motivating work will be 2.5D formulations of matrix multiplication and LU factorization for dense linear algebra [37]. These algorithms are an adaptation and generalization of 3D matrix multiplication [15], [2], [3], [7], [26]. The main idea is to store redundant intermediate data, in order to reduce communication bandwidth. Bandwidth is reduced by a factor of $\sqrt{c}$ at the cost of a memory usage overhead of a factor of c. The technique is particularly useful for the strong scaling regime, where one can solve problems faster by storing more intermediates spread over more processors.

III. Divide-and-Conquer APSP

The all-pairs shortest-paths problem corresponds to finding the matrix closure on the tropical $(\min, +)$ semiring. A semiring is denoted by $(\mathbb{S}, \oplus, \otimes, 0, 1)$, where $\oplus$ and $\otimes$ are binary operations defined on the set $\mathbb{S}$ with identity elements 0 and 1, respectively [17]. In the case of the *tropical semiring*, $\oplus$ is $\min$, $\otimes$ is $+$, the additive identity is $+\infty$, and the multiplicative identity is 0. Compared to the classical matrix multiplication over the ring of real numbers, in our semiring-matrix-matrix multiplication (also called the distance product [44]), each multiply operation is replaced with an addition (to calculate the length of a larger path from smaller paths or edges) and each add operation is replaced with a minimum operation (to get the minimum in the presence of multiple paths).

Algorithm 1 gives the high-level structure of the divide-and-conquer all-pairs-shortest-path algorithm (DC-APSP). The workflow of the DC-APSP algorithm is also pictured in Figure 2. The correctness of this algorithm has been proved by many researchers [4], [10], [32] using various methods. Edge weights can be arbitrary, including negative numbers, but we assume that the graph is free of negative cycles. The tropical semiring does not have additive inverses, hence fast matrix multiplication algorithms like those by Strassen [38] and Coppersmith-Winograd [13] are not applicable for this problem.

For simplicity, we formulate our algorithms and give results only for adjacency matrices of power-of-two dimension. Extending the algorithms and analysis to general adjacency matrices is straight-forward.

Each semiring-matrix-matrix multiplication performs $O(n^3)$ additions and $O(n^2)$ minimum (min) operations. If we count each addition and min operation as $O(1)$ flops, the total computation cost of DC-APSP, F, is given by a recurrence

$$F(n) = 2 \cdot F(n/2) + O(n^3) = O(n^3).$$

Thus the number of operations is the same as that required for matrix multiplication.

IV. Communication lower bounds

A good parallel algorithm has as little inter-processor communication as possible. In this section, we prove lower bounds on the inter-processor communication required to compute DC-APSP in parallel. All of our lower bounds are extensions of dense linear algebra communication lower bounds.

A. Bandwidth lower bound

We measure the bandwidth cost as the number of words (bytes) sent or received by any processor along the critical path of execution. Semiring matrix multiplication has the same computational dependency structure as classical matrix multiplication. The same communication cost analysis applies because only the scalar multiply and add operations are different. Our analysis will assume no data is replicated at the start and that the computational work is load-balanced.

The lower bound on bandwidth cost of matrix multiplication is due to Hong and Kung [22], [23]. Ballard et al. [6] extended those lower bounds to other traditional numerical linear algebra algorithms. For a local memory of size M, matrix multiplication requires

$$W(M) = \Omega\left(\frac{n^3}{p\sqrt{M}}\right) \tag{1}$$

words to be sent by some processor. Further, for a memory of any size, matrix multiplication requires

$$W = \Omega\left(\frac{n^2}{p^{2/3}}\right)$$

words to be sent [3], [23], [37]. These bounds apply directly to semiring matrix multiplication and consequently to DC-APSP, which performs many semiring matrix multiplications.

B. Latency lower bound

The first bandwidth lower-bound in the previous section (Equation 1), provides a latency lower-bound on semiring-matrix multiplication. Since no message can be larger than the local memory on a given processor,

$$S(M) = \Omega\left(\frac{n^3}{p \cdot M^{3/2}}\right)$$

A = DC-APSP(A, n)
// Input: $A \in \mathbb{S}^{n \times n}$ is a graph adjacency matrix of a n-node graph G
// Output: $A \in \mathbb{S}^{n \times n}$ is the APSP distance matrix of G
1 **if** n == 1
2 return.
Partition $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, where all A_{ij} are $n/2$-by-$n/2$ // Partition the vertices $V = (V_1, V_2)$
3 A_{11} = DC-APSP$(A_{11}, n/2)$ // Find all-pairs shortest paths between vertices in V_1
4 $A_{12} = A_{11} \cdot A_{12}$ // Propogate paths from V_1 to V_2
5 $A_{21} = A_{21} \cdot A_{11}$ // Propogate paths from V_2 to V_1
6 $A_{22} = \min(A_{22}, A_{21} \cdot A_{12})$ // Update paths to V_2 via paths from V_2 to V_1 and back to V_2
7 A_{22} = DC-APSP$(A_{22}, n/2)$ // Find all-pairs shortest paths between vertices in V_2
8 $A_{21} = A_{22} \cdot A_{21}$ // Find shortest paths from V_2 to V_1
9 $A_{12} = A_{12} \cdot A_{22}$ // Find shortest paths from V_1 to V_2
10 $A_{11} = \min(A_{11}, A_{12} \cdot A_{21})$ // Find all-pairs shortest paths for vertices in V_1

Fig. 1. A divide-and-conquer algorithm for the all-pairs shortest-paths problem

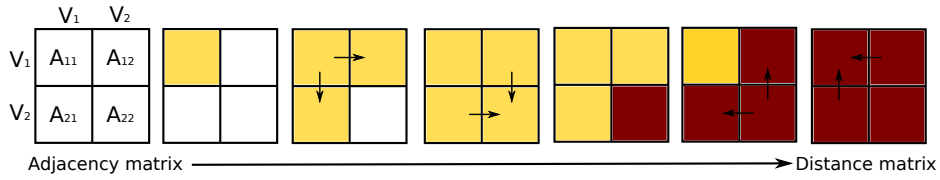


Fig. 2. DC-APSP algorithm, with initial adjacency distance denoted in white, partially complete path distances in yellow, and final path distances in red

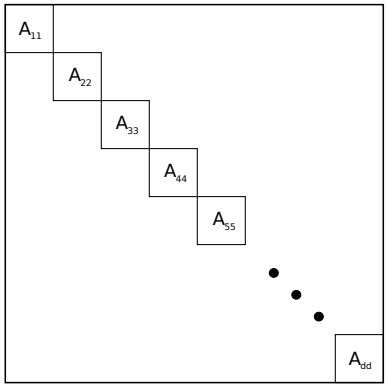


Fig. 3. DC-APSP diagonal block dependency path. These blocks must be computed in order and communication is required between each block.

messages must be sent by some processor. This latency lower-bound applies for classical and semiring matrix multiplication, as well as DC-APSP.

However, we can obtain a tighter lower-bound for DC-APSP by considering the dependency structure of the algorithm. As it turns out, we can use the same argument as presented in [37] for 2.5D LU factorization. Figure 3 considers how the distance matrix A is blocked along its diagonal.

We assume all blocks are of the same size, and adjacent blocks belong to different processors. If a process owns a block of dimension b, we assume the process computes all b^2 entries of the distance matrix and no other process computes these entries (no recomputation). Finally, we assume each block is computed sequentially. Now, we see that between the computations of two diagonal blocks, $\Omega(1)$ message and $\Omega(b^2)$ words must move between the two processors computing the diagonal blocks. This holds since the shortest paths going to a future diagonal block of the distance matrix may include any of the shortest paths computed in previous diagonal blocks.

These requirements yield a lower bound on the latency cost. If we desire a bandwidth cost of

$$W = \Omega(d \cdot (n/d)^2) = \Omega(n^2/d) = \Omega\left(\frac{n^2}{\sqrt{cp}}\right),$$

for some c, we must incur a latency cost of

$$S = \Omega(d) = \Omega(\sqrt{cp}).$$

More generally, we have

$$S \cdot W = \Omega(n^2).$$

We conjecture that this lower-bound holds with looser assumptions, in particular for any data layout. We are working

C = 2D-SMMM$(A, B, C, \Lambda[1:\sqrt{p}, 1:\sqrt{p}], n, p)$
// Input: process $\Lambda[i,j]$ owns $A_{ij}, B_{ij}, C_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}}$
// Output: process $\Lambda[i,j]$ owns $C_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}}$
1 **parallel for** $i, j = 1$ **to** $\sqrt{p}$
2 **for** $k = 1$ **to** $\sqrt{p}$
3 Broadcast A_{ik} to processor columns $\Lambda[i, :]$
4 Broadcast B_{kj} to processor rows $\Lambda[:, j]$
5 $C_{ij} = \min(C_{ij}, A_{ik} \cdot B_{kj})$

Fig. 4. An algorithm for Semiring-matrix-matrix multiplication on a 2D processor grid.

on generalizing the argument to reason with respect to the computational depenency graph instead.

V. Parallelization of DC-APSP

In this section, we introduce techniques for parallelization of the divide-and-conquer all-pairs-shortest-path algorithm (DC-APSP). Our first approach uses a 2D block-cyclic parallelization. We demonstrate that a careful choice of block-size can minimize both latency and bandwidth costs simultaneously. Our second approach utilizes a 2.5D decomposition [35], [37]. Our cost analysis shows that the 2.5D algorithm reduces the bandwidth cost and improves strong scalability.

A. 2D Divide-and-Conquer APSP

We start by deriving a parallel DC-APSP algorithm that operates on a square 2D processor grid and consider cyclic and blocked variants.

1) 2D Semiring-Matrix-Matrix-Multiply: Algorithm 4 describes an algorithm for performing Semiring-Matrix-Matrix-Multiply (SMMM) on a 2D processor grid denoted by Λ. Since the data dependency structure of SMMM is identical to traditional matrix multiply, we employ the popular SUMMA algorithm [42]. The algorithm is formulated in terms of distributed rank-1 updates. These updates are associative and commutative so they can be pipelined or blocked. To achieve optimal communication performance, the matrices should be laid out in a blocked fashion, and each row and column of processors should broadcast its block-row and block-column in turn. Given p processors, each processor would then receive $O(\sqrt{p})$ messages of size $O(n^2/p)$, giving a bandwidth cost of $O(n^2/\sqrt{p})$. We note that any different classical distributed matrix multiplication algorithm (e.g. Cannon's algorithm [11]) can be used here in place of SUMMA.

2) 2D blocked Divide-and-Conquer APSP: Algorithm 8 (psuedo-code given in the Appendix) displays a parallel 2D blocked version of the DC-APSP algorithm. In this algorithm, each SMMM is computed on the quadrant of the processor grid on which the result belongs. The operands, A and B, must be sent to the processor grid quadrant on which C is distributed. At each recursive step, the algorithm recurses into one quadrant of the processor grid. Similar to SMMM, this is also an owner computes algorithm in the sense that the processor that owns the submatrix to be updated does the computation itself after receiving required inputs from other processors.

This blocked algorithm has a clear flaw, in that at most a quarter of the processors are active at any point in the algorithm. We will alleviate this load-imbalance by introducing a block-cyclic version of the algorithm.

3) 2D block-cyclic Divide-and-Conquer APSP: Algorithm 9 (given in the Appendix) details the full 2D block-cyclic DC-APSP algorithm. This block-cyclic algorithm operates by performing *cyclic-steps* until a given block-size, then proceeding with *blocked-steps* by calling the blocked algorithm as a subroutine. At each cyclic-step, each processor operates on sub-blocks of its local block, while at each blocked-step a sub-grid of processors operate on their full matrix blocks. In other words, a cyclic-step reduces the local working sets, while a blocked-step reduces the number of active processors. These two steps are demonstrated in sequence in Figure 5 with 16 processors.

We note that no redistribution of data is required to use a block-cyclic layout. Traditionally, (e.g. in ScaLAPACK [8]) using a block-cyclic layout requires that each processor own a block-cyclic portion of the starting matrix. However, the APSP problem is invariant to permutation (permuting the numbering of the node labels does not change the answer). We exploit permutation invariance by assigning each process the same sub-block of the adjacency and distance matrices, no matter how many blocked or cyclic steps are taken.

As derived in Appendix A in [36], if the block size is picked as $b = O(n/\log(p))$ (execute $O(\log\log(p))$ cyclic recursive steps), the bandwidth and latency costs are

$$W_{\text{bc-2D}}(n,p) = O(n^2/\sqrt{p}),$$
$$S_{\text{bc-2D}}(p) = O(\sqrt{p}\log^2(p)).$$

These costs are optimal (modulo the polylog latency term) when the memory size is $M = O(n^2/p)$. The costs are measured along the critical path of the algorithm, showing that both the computation and communication are load balanced throughout execution.

B. 2.5D DC-APSP

In order to construct a communication-optimal DC-APSP algorithm, we utilize 2.5D-SMMM. Transforming 2D SUMMA (Algorithm 4) to a 2.5D algorithm can be done by performing a different subset of updates on each one of c processor layers. Algorithm 10 (given in the Appendix) details 2.5D SUMMA, modified to perform SMMM. The three dimensional processor grids used in 2.5D algorithms are denoted by Π.

Given a replication factor $c \in [1, p^{1/3}]$, each $\sqrt{p/c}$-by-$\sqrt{p/c}$ processor layer performs n/c outer products. Since each length n outer product vector is subdivided into $\sqrt{p/c}$ chunks, the bandwidth cost is $O(n^2/\sqrt{cp})$ words. These outer products

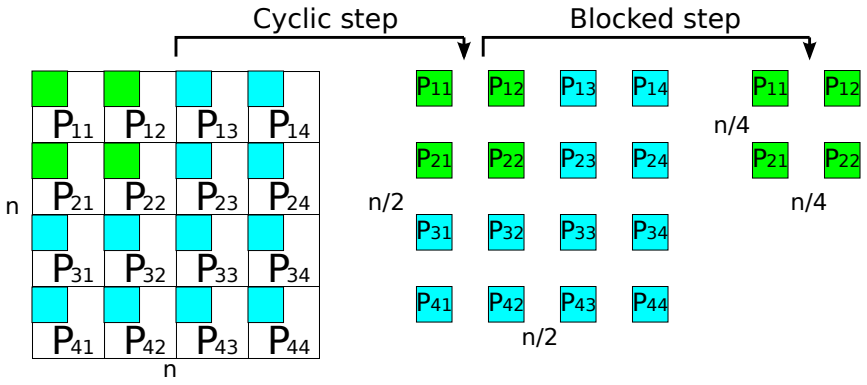


Fig. 5. Our block-cyclic 2D APSP algorithm performs cyclic-steps until a given block-size, then performs blocked-steps as shown in this diagram.

can be blocked into bundles of up to $n/\sqrt{p/c}$ to lower the latency cost to $O(\sqrt{p/c^3})$ messages.

Algorithm 11 (pseudo-code given in the Appendix) displays the blocked version of the 2.5D DC-APSP algorithm. The blocked algorithm executes multiplies and recurses on octants of the processor grid (rather than quadrants in the 2D version). The algorithm recurses until $c = 1$, which must occur while $p \geq 1$, since $c \leq p^{1/3}$. The algorithm then calls the 2D block-cyclic algorithm on the remaining 2D sub-partition.

The 2.5D blocked algorithm suffers from load-imbalance. In fact, the top half of the processor grid does no work. We can fix this by constructing a block-cyclic version of the algorithm, which performs cyclic steps with the entire 3D processor grid, until the block-size is small enough to switch to the blocked version. The 2.5D block-cyclic algorithm looks exactly like Algorithm 9, except each call to 2D SMMM is replaced with 2.5D SMMM. This algorithm is given in full in [36].

As derived in Appendix B in [36], if the 2.5D block size is picked as $b_1 = O(n/c)$ (execute $O(\log(c))$ 2.5D cyclic recursive steps), the bandwidth and latency costs are

$$W_{\text{bc-2.5D}}(n,p) = O(n^2/\sqrt{cp}),$$
$$S_{\text{bc-2.5D}}(p) = O(\sqrt{cp}\log^2(p)).$$

These costs are optimal for any memory size (modulo the polylog latency term).

VI. Experiments

In this section, we show that the distributed APSP algorithms do not just lower the theoretical communication cost, but actually improve performance on large supercomputers. We implement the 2D and 2.5D variants of DC-APSP recursively, as described in the previous section. For fairness, both variants have the same amount of optimizations applied and use the same kernel. We were not able to find any publicly available distributed memory implementations of APSP for comparison.

A. Implementation

The dominant sequential computational work of the DC-APSP algorithm is the Semiring-Matrix-Matrix-Multiplies (SMMM) called at every step of recursion. Our implementation of SMMM uses two-level cache-blocking, register blocking, explicit SIMD intrinsics, and loop unrolling. We implement threading by assigning L1-cache blocks of C to different threads.

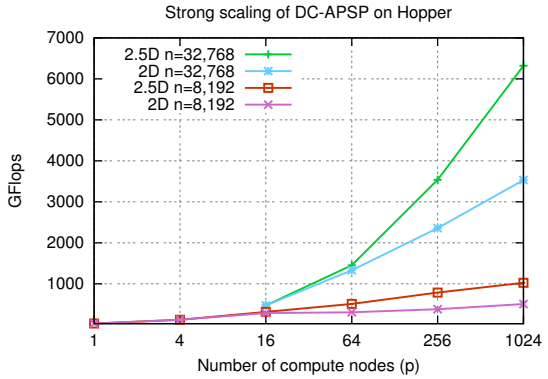
Our 2.5D DC-APSP implementation generalizes the following algorithms: 2D cyclic, 2D blocked, 2D block-cyclic, 2.5D blocked, 2.5D cyclic, and 2.5D block-cyclic. Block sizes b_1 and b_2 control how many 2.5D and 2D cyclic and blocked steps are taken. These block-sizes are set at run-time and require no modification to the algorithm input or distribution.

We compiled our codes with the GNU C/C++ compilers (v4.6) with the -O3 flag. We use Cray's MPI implementation, which is based on MPICH2. We run 4 MPI processes per node, and use 6-way intra-node threading with the GNU OpenMP library. The input is an adjacency matrix with entries representing edge-weights in double-precision floating-point numbers.

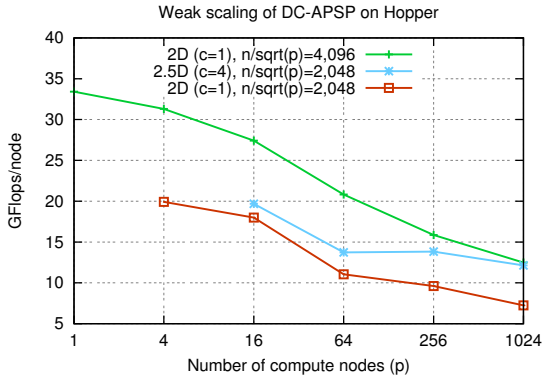
B. Performance

Our experimental platform is 'Hopper', which is a Cray XE6 supercomputer, built from dual-socket 12-core "Magny-Cours" Opteron compute nodes. Each node can be viewed as a four-chip compute configuration due to NUMA domains. Each of these four chips have six super-scalar, out-of-order cores running at 2.1 GHz with private 64 KB L1 and 512 KB L2 caches. The six cores on a chip share a 6 MB L3 cache and dual DDR3-1333 memory controllers capable of providing an average stream [30] bandwidth of 12 GB/s per chip. Nodes are connected through Cray's 'Gemini' network, which has a 3D torus topology. Each Gemini chip, which is shared by two Hopper nodes, is capable of 9.8 GB/s bandwidth.

Our threaded Semiring-Matrix-Matrix-Multiply achieves up to 13.6 GF on 6-cores of Hopper, which is roughly 25% of theoretical floating-point peak. This is a fairly good fraction in the absence of an equivalent fused multiply-add operation for our semiring. Our implementation of DC-APSP uses this subroutine to perform APSP at 17% of peak computational performance on 1 node (24 cores, 4 processes, 6 threads per process).

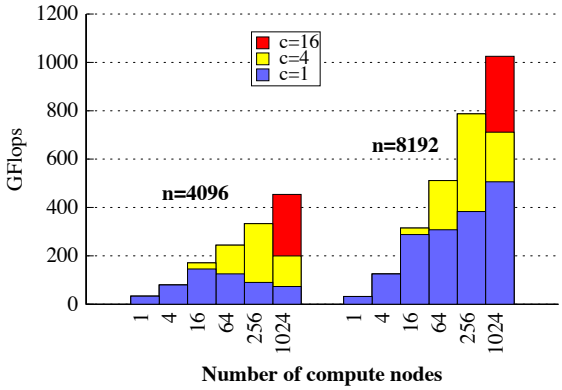


(a) DC-APSP strong scaling

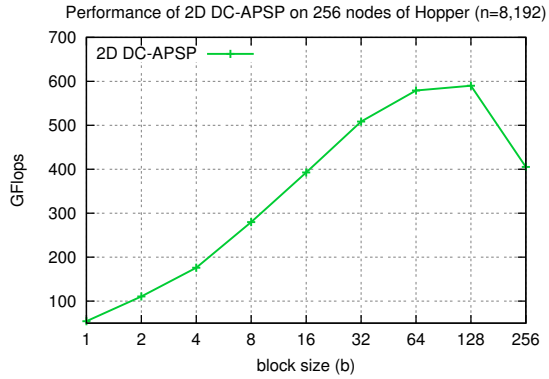


(b) DC-APSP weak scaling

Fig. 6. Scaling of 2D and 2.5D block-cyclic DC-APSP on Hopper (Cray XE6)



(a) 2.5D DC-APSP small matrix performance



(b) Performance of DC-APSP with respect to block size

Fig. 7. Performance of DC-APSP on small matrices

Figure 6(a) demonstrates the strong scaling performance of 2D and 2.5D APSP. Strong scaling performance is collected by keeping the adjacency matrix size constant and computing APSP with more processors. The 2.5D performance is given as the best performing variant for any replication factor c (in almost all cases, $c = 4$). Strong scaling a problem to a higher core-count lowers the memory usage per processor, allowing increased replication (increased c). Performing 2.5D style replication improves efficiency significantly, especially at large scale. On 24,576 cores of Hopper, the 2.5D algorithm improves on the performance of the 2D APSP algorithm by a factor of 1.8x for $n = 8,192$ and 2.0x for $n = 32,768$.

Figure 6(b) shows the weak scaling performance of the 2D and 2.5D DC-APSP algorithms. To collect weak scaling data, we keep the problem size per processor $(n/\sqrt{p})$ constant and grow the number of processors. Since the memory usage per processor does not decrease with the number of processors during weak scaling, the replication factor cannot increase. We compare data with $n/\sqrt{p} = 2048, 4096$ for 2D ($c = 1$) and with $n/\sqrt{p} = 2048$ for 2.5D ($c = 4$). The 2.5D DC-APSP algorithm performs almost as well as the 2D algorithm with a larger problem size and significantly better than the 2D algorithm with the same problem size.

The overall weak-scaling efficiency is good all the way up to the 24,576 cores (1024 nodes), where the code achieves an impressive aggregate performance over 12 Teraflops (Figure 6(b)). At this scale, our 2.5D implementation solves the all-pairs shortest-paths problem for 65,536 vertices in roughly 2 minutes. With respect to 1-node performance, strong scaling allows us to solve a problem with 8,192 vertices over 30x faster on 1024 compute nodes (Figure 7(a)). Weak scaling gives us a performance rate up to 380x higher on 1024 compute nodes than on one node.

Figure 7(a) shows the performance of 2.5D DC-APSP on small matrices. The bars are stacked so the $c = 4$ case shows the added performance over the $c = 1$ case, while the $c = 16$ case shows the added performance over the $c = 4$ case. A replication factor of $c = 16$ results in a speed-up of 6.2x for the smallest matrix size $n = 4,096$. Overall, we see that 2.5D algorithm hits the scalability limit much later than the 2D counterpart. Tuning over the block sizes (Figure 7(b)), we also see the benefit of the block-cyclic layout for the 2D algorithm. The best performance over all block sizes is significantly higher than either the cyclic ($b = 1$) or blocked ($b = n/\sqrt{p}$) performance. We found that the use of cyclicity in the 2.5D algorithm supplanted the need for cyclic steps in the nested call to the 2D algorithm. The 2.5D blocked algorithm can call the 2D blocked algorithm directly without a noticeable performance loss.

VII. Discussion of alternatives

We solved the APSP problem using a distributed memory algorithm that minimizes communication and maximizes temporal locality reuse through BLAS-3 subroutines. There are at least two other alternatives to solving this problem. One alternative is to use a sparse APSP algorithm and the other one is to leverage an accelerator architecture such as GPU.

If the graph is big enough so that it requires distribution to multiple processors, the performance of sparse APSP algorithms become heavily dependent on the structure of the graph; and rather poor in general. For the case that the graph is small enough so that it can be fully replicated along different processors, one can parallelize Johnson's algorithm in an embrarrassingly parallel way. We experimented with this case, where each core runs many to all shortest paths. Specifically, we wanted to know how sparse the graph needs to get in order to make this fully replicated approach a strong alternative. The breakeven points for density depend both on the graph size (the number of vertices) and the the number of cores. For example, using 384 cores, solving the APSP problem on a 16,384 vertex, 5% dense graph, is slightly faster using our approach (18.6 vs. 22.6 seconds) than using the replicated Johnson's algorithm. Keeping the number of vertices intact and further densifying the graph favors our algorithm while sparsifying it favors Johnson's algorithm. Larger cores counts also favor Johnson's algorithm; but its major disadvantage is its inability to run any larger problems due to graph replication.

On the architecture front, we benchmarked a highly optimized CUDA implementation [10] on a single Fermi (NVIDIA X2090) GPU. This GPU implementation also runs the dense recursive algorithm described in this paper. On a graph with 8,192 vertices, our distributed memory CPU based implementation running on 4 nodes achieved 80% of the performance of the Fermi (which takes 9.9 seconds to solve APSP on this graph). This result shows the suitability of the GPU architure to the APSP problem, and provides us a great avenue to explore as future work. As more supercomputers become equipped with GPU accelerators, we plan to reimplement our 2.5D algorithm in a way that it can take advantage of the GPUs as coprocessors on each node. The effect of communication avoidance will become more pronounced as local compute phases get faster due to GPU acceleration.

VIII. Conclusion

The divide-and-conquer APSP algorithm is well suited for parallelization in a distributed memory environment. The algorithm resembles well-studied linear algebra algorithms (e.g. matrix multiply, LU factorization). We exploit this resemblance to transfer implementation and optimization techniques from the linear algebra domain to the graph-theoretic APSP problem. In particular, we use a block-cyclic layout to load-balance the computation and data movement, while simultaneously minimizing message latency overhead. Further, we formulate a 2.5D DC-APSP algorithm, which lowers the bandwidth cost and improves parallel scalability. Our implementations of these algorithms achieve good scalability at very high concurrency and confirm the practicality of our analysis. Our algorithm provides a highly parallel solution to the decrease-only version of the metric nearness problem problem as well, which is equivalent to APSP.

Our techniques for avoiding communication allow for a scalable implementation of the divide-and-conquer APSP algorithm. The benefit of such optimizations grows with machine size and level of concurrency. The performance of our implementation can be further improved upon by exploiting locality via topology-aware mapping. The current Hopper job scheduler does not allocate contiguous partitions but other supercomputers (e.g. IBM BlueGene) allocate toroidal partitions, well-suited for mapping of 2D and 2.5D algorithms [35].

IX. Acknowledgements

The first author was supported by a Krell Department of Energy Computational Science Graduate Fellowship, grant number DE-FG02-97ER25308. The second author was supported by the Director, Office of Science, U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We acknowledge funding from Microsoft (Award #024263) and Intel (Award #024894), and matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from ParLab affiliates National Instruments, Nokia, NVIDIA, Oracle and Samsung, as well as MathWorks. Research is also supported by DOE grants DE-SC0004938, DE-SC0005136, DE-SC0003959, DE-SC0008700, and AC02-05CH11231, and DARPA grant HR0011-12-2-0016. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] Hopper, NERSC's Cray XE6 system. http://www.nersc.gov/users/computational-systems/hopper/.

[2] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM J. Res. Dev.*, 39:575–582, September 1995.

[3] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3 – 28, 1990.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman, Boston, MA, USA, 1974.

[5] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication: regular submission. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 1–12, New York, NY, USA, 2011. ACM.

[6] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications*, 32(3):866–901, 2011.

[7] J. Berntsen. Communication efficient matrix multiplication on hypercubes. *Parallel Computing*, 12(3):335 – 342, 1989.

[8] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[9] J. Brickell, I. S. Dhillon, S. Sra, and J. A. Tropp. The metric nearness problem. *SIAM J. Matrix Anal. Appl.*, 30:375–396, 2008.

[10] A. Buluç, J. R. Gilbert, and C. Budak. Solving path problems on the GPU. *Parallel Computing*, 36(5-6):241 – 253, 2010.

[11] L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Bozeman, MT, USA, 1969.

[12] R. A. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. In *Proceedings of the nineteenth annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 71–80, New York, NY, USA, 2007. ACM.

[13] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the Nineteenth Annual ACM Conference on Theory of Computing*, pages 1–6, New York, NY, USA, 1987. ACM Press.

[14] A. R. Curtis, T. Carpenter, M. Elsheikh, A. LÃşpez-Ortiz, and S. Keshav. REWIRE: an optimization-based framework for data center network design. In *INFOCOM*, 2012.

[15] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM Journal on Computing*, 10(4):657–675, 1981.

[16] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. Phast: Hardware-accelerated shortest path trees. *Parallel and Distributed Processing Symposium, International*, 0:921–931, 2011.

[17] J. G. Fletcher. A more general algorithm for computing closed semiring costs between vertices of a directed graph. *Communications of the ACM*, 23(6):350–351, 1980.

[18] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5:345–, June 1962.

[19] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th international conference on Experimental algorithms*, WEA'08, pages 319–333, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] M. B. Habbal, H. N. Koutsopoulos, and S. R. Lerman. A decomposition algorithm for the all-pairs shortest path problem on massively parallel computer architectures. *Transportation Science*, 28(4):292–308, 1994.

[21] S. Holzer and R. Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 355–364. ACM, 2012.

[22] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 326–333, New York, NY, USA, 1981. ACM.

[23] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[24] J. Jenq and S. Sahni. All pairs shortest paths on a hypercube multiprocessor. In *ICPP '87: Proc. of the Intl. Conf. on Parallel Processing*, pages 713–716, 1987.

[25] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[26] S. L. Johnsson. Minimizing the communication time for matrix multiplication on multiprocessors. *Parallel Comput.*, 19:1235–1257, November 1993.

[27] V. Kumar and V. Singh. Scalability of parallel algorithms for the all-pairs shortest-path problem. *J. Parallel Distrib. Comput.*, 13:124–138, 1991.

[28] R. C. Larson and A. R. Odoni. *Urban operations research*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1981.

[29] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM J. Numer. Analysis*, 16:346–358, 1979.

[30] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

[31] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[32] J.-S. Park, M. Penner, and V. K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):769–782, 2004.

[33] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination. In *Proceedings of seventh annual ACM Symposium on Theory of Computing*, STOC '75, pages 245–254, New York, NY, USA, 1975. ACM.

[34] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

[35] E. Solomonik, A. Bhatele, and J. Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *Supercomputing, Seattle, WA, USA*, Nov 2011.

[36] E. Solomonik, A. Buluç, and J. Demmel. Minimizing communication in all-pairs shortest-paths. Technical Report UCB/EECS-2012-19, EECS Department, University of California, Berkeley, Feb 2012.

[37] E. Solomonik and J. Demmel. Communication-optimal 2.5D matrix multiplication and LU factorization algorithms. In *Lecture Notes in Computer Science, Euro-Par, Bordeaux, France*, Aug 2011.

[38] V. Strassen. Gaussian elimination is not optimal. *Numerical Math.*, 13:354–356, 1969.

[39] A. Tiskin. All-pairs shortest paths computation in the bsp model. In F. Orejas, P. Spirakis, and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 178–189. Springer Berlin / Heidelberg, 2001.

[40] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal of Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[41] J. D. Ullman and M. Yannakakis. High probability parallel transitive-closure algorithms. *SIAM Journal of Computing*, 20:100–125, February 1991.

[42] R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[43] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9:11–12, January 1962.

[44] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, May 2002.

APPENDIX
DETAILED PSEUDOCODES

A = BLOCKED-DC-APSP$(A, \Lambda[1 : \sqrt{p}, 1 : \sqrt{p}], n, p)$
  // Input: process $\Lambda[i,j]$ owns a block of the adjacency matrix, $A_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}}$
  // Output: process $\Lambda[i,j]$ owns a block of the APSP distance matrix, $A_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}}$
1 **if** p == 1
2 A = DC-APSP(A, n)
  // Partition the vertices $V = (V_1, V_2)$ by partitioning the processor grid
  Partition $\Lambda = \begin{bmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{21} & \Lambda_{22} \end{bmatrix}$, where all Λ_{ij} are $\sqrt{p}/2$-by-$\sqrt{p}/2$
3 **parallel for** $i, j = 1$ **to** $\sqrt{p}/2$
    // Find all-pairs shortest paths between vertices in V_1
4 A_{ij} = BLOCKED-DC-APSP$(A_{ij}, \Lambda_{11}, n/2, p/4)$
    // Propogate paths from V_1 to V_2
5 $\Lambda_{11}[i,j]$ sends A_{ij} to $\Lambda_{12}[i,j]$.
6 $A_{i,j+\sqrt{p}/2}$ = 2D-SMMM$(A_{ij}, A_{i,j+\sqrt{p}/2}, A_{i,j+\sqrt{p}/2}, \Lambda_{12}, n/2, p/4)$
    // Propogate paths from V_2 to V_1
7 $\Lambda_{11}[i,j]$ sends A_{ij} to $\Lambda_{21}[i,j]$.
8 $A_{i+\sqrt{p}/2,j}$ = 2D-SMMM$(A_{i+\sqrt{p}/2,j}, A_{ij}, A_{i+\sqrt{p}/2,j}, \Lambda_{21}, n/2, p/4)$
    // Update paths to V_2 via paths from V_2 to V_1 and back to V_2
9 $\Lambda_{12}[i,j]$ sends $A_{i,j+\sqrt{p}/2}$ to $\Lambda_{22}[i,j]$.
10 $\Lambda_{21}[i,j]$ sends $A_{i+\sqrt{p}/2,j}$ to $\Lambda_{22}[i,j]$.
11 $A_{i+\sqrt{p}/2,j+\sqrt{p}/2}$ = 2D-SMMM$(A_{i+\sqrt{p}/2,j}, A_{i,j+\sqrt{p}/2}, A_{i+\sqrt{p}/2,j+\sqrt{p}/2}, \Lambda_{22}, n/2, p/4)$
    // Find all-pairs shortest paths between vertices in V_2
12 $A_{i+\sqrt{p}/2,j+\sqrt{p}/2}$ = BLOCKED-DC-APSP$(A_{i+\sqrt{p}/2,j+\sqrt{p}/2}, \Lambda_{22}, n/2, p/4)$
    // Find shortest paths paths from V_2 to V_1
13 $\Lambda_{22}[i,j]$ sends $A_{i+\sqrt{p}/2,j+\sqrt{p}/2}$ to $\Lambda_{21}[i,j]$.
14 $A_{i+\sqrt{p}/2,j}$ = 2D-SMMM$(A_{i+\sqrt{p}/2,j+\sqrt{p}/2}, A_{i+\sqrt{p}/2,j}, A_{i+\sqrt{p}/2,j}, \Lambda_{21}, n/2, p/4)$
    // Find shortest paths paths from V_1 to V_2
15 $\Lambda_{22}[i,j]$ sends $A_{i+\sqrt{p}/2,j+\sqrt{p}/2}$ to $\Lambda_{12}[i,j]$.
16 $A_{i,j+\sqrt{p}/2}$ = 2D-SMMM$(A_{i,j+\sqrt{p}/2}, A_{i+\sqrt{p}/2,j+\sqrt{p}/2}, A_{i,j+\sqrt{p}/2}, \Lambda_{12}, n/2, p/4)$
    // Find all-pairs shortest paths for vertices in V_1
17 $\Lambda_{12}[i,j]$ sends $A_{i,j+\sqrt{p}/2}$ to $\Lambda_{11}[i,j]$.
18 $\Lambda_{21}[i,j]$ sends $A_{i+\sqrt{p}/2,j}$ to $\Lambda_{11}[i,j]$.
19 A_{ij} = 2D-SMMM$(A_{i,j+\sqrt{p}/2}, A_{i+\sqrt{p}/2,j}, A_{ij}, \Lambda_{22}, n/2, p/4)$

Fig. 8. A blocked parallel divide-and-conquer algorithm for the all-pairs shortest-paths problem

A = BLOCK-CYCLIC-DC-APSP$(A, \Lambda[1 : \sqrt{p}, 1 : \sqrt{p}], n, p, b)$

// On input, process $\Lambda[i,j]$ owns a block of the adjacency matrix, $A_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}}$

// On output, process $\Lambda[i,j]$ owns a block of the APSP distance matrix, $A_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}}$

if $n \leq b$

  A = BLOCKED-DC-APSP(A, Λ, n, p) // Switch to blocked algorithm once the matrix is small

parallel for $i, j = 1$ **to** $\sqrt{p}$

  $A^l = A_{ij}$ // A^l denotes the local matrix owned by $\Lambda[i,j]$

  Partition $A^l = \begin{bmatrix} A^l_{11} & A^l_{12} \\ A^l_{21} & A^l_{22} \end{bmatrix}$, where all A^l_{kl} are $n/2$-by-$n/2$ // Partition the vertices $V = (V_1, V_2)$

  A^l_{11} = BLOCK-CYLIC-DC-APSP$(A^l_{11}, \Lambda, n/2, p, b)$ // Find all-pairs shortest paths between vertices in V_1

  A^l_{12} = 2D-SMMM$(A^l_{11}, A^l_{12}, A^l_{12}, \Lambda, n/2, p)$ // Propogate paths from V_1 to V_2

  A^l_{21} = 2D-SMMM$(A^l_{21}, A^l_{11}, A^l_{21}, \Lambda, n/2, p)$ // Propogate paths from V_2 to V_1

  A^l_{22} = 2D-SMMM$(A^l_{21}, A^l_{12}, A^l_{22}, \Lambda, n/2, p)$ // Update paths among vertices in V_2 which go through V_1

  A^l_{22} = BLOCK-CYLIC-DC-APSP$(A^l_{22}, \Lambda, n/2, p, b)$ // Find all-pairs shortest paths between vertices in V_2

  A^l_{21} = 2D-SMMM$(A^l_{22}, A^l_{21}, A^l_{21}, \Lambda, n/2, p)$ // Find shortest paths from V_2 to V_1

  A^l_{12} = 2D-SMMM$(A^l_{12}, A^l_{22}, A^l_{12}, \Lambda, n/2, p)$ // Find shortest paths from V_1 to V_2

  A^l_{11} = 2D-SMMM$(A^l_{12}, A^l_{21}, A^l_{11}, \Lambda, n/2, p)$ // Find all-pairs shortest paths for vertices in V_1

Fig. 9. A block-cyclic parallel divide-and-conquer algorithm for the all-pairs shortest-paths problem

C = 2.5D-SMMM$(A, B, C, \Pi[1 : \sqrt{p/c}, 1 : \sqrt{p/c}, 1 : c], n, p, c)$

// Input: process $\Pi[i,j,1]$ owns $A_{ij}, B_{ij}, C_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}}}$

// Output: process $\Pi[i,j,1]$ owns $C_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}}}$

parallel for $m = 1$ **to** c

  parallel for $i, j = 1$ **to** $\sqrt{p}$

    Π_{ij1} sends A_{ij} to process $\Pi_{i,j,j/c}$

    Π_{ij1} sends B_{ij} to process $\Pi_{i,j,i/c}$

    if m == 1

      $C_{ijm} = C_{ij}$

    else $C_{ijm}[:, :] = \infty$

      for $k = 1$ **to** $\sqrt{p/c^3}$

        Broadcast $A_{i,m\sqrt{p/c^3}+k}$ to processor columns $\Pi[i, :, m]$

        Broadcast $B_{m\sqrt{p/c^3}+k,j}$ to processor rows $\Pi[:, j, m]$

        $C_{ijm} = \min(C_{ij}, A_{i,m\sqrt{p/c^3}+k} \cdot B_{m\sqrt{p/c^3}+k,j})$

      Reduce to first processor layer, $C_{ij} = \sum_{m=1}^{c} C_{ijm}$

Fig. 10. An algorithm for Semiring-matrix-matrix multiplication on a 2D processor grid.

A = 2.5D-BLOCKED-DC-APSP$(A, \Pi[1 : \sqrt{p/c}, 1 : \sqrt{p/c}, 1 : c], n, p, c, b)$

// Input: process $\Pi[i, j, 1]$ owns a block of the adjacency matrix, $A_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}}}$

// Output: process $\Pi[i, j, 1]$ owns a block of the APSP distance matrix, $A_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}}}$

1 **if** c == 1

2 A = BLOCK-CYCLIC-DC-APSP(A, n, p, c, b)

// Partition the vertices $V = (V_1, V_2)$ by partitioning the processor grid

Partition Π into 8 cubic block Π_{abc}, for $a, b, c \in \{1, 2\}$, where all Π_{abc} are $\sqrt{p/c}/2$-by-$\sqrt{p/c}/2$-by-$c/2$

3 **parallel for** $k = 1$ **to** $c/2$

4 **parallel for** $i, j = 1$ **to** $\sqrt{p}/2$

// Find all-pairs shortest paths between vertices in V_1

5 A_{ij} = 2.5D-BLOCKED-DC-APSP$(A_{ij}, \Pi_{111}, n/2, p/8)$

// Propogate paths from V_1 to V_2

6 $\Pi_{111}[i, j]$ sends A_{ij} to $\Pi_{121}[i, j]$.

7 $A_{i,j+\sqrt{p/c}/2}$ = 2.5D-SMMM$(A_{ij}, A_{i,j+\sqrt{p/c}/2}, A_{i,j+\sqrt{p/c}/2}, \Pi_{121}, n/2, p/8, c/2)$

// Propogate paths from V_2 to V_1

8 $\Pi_{111}[i, j]$ sends A_{ij} to $\Pi_{211}[i, j]$.

9 $A_{i+\sqrt{p}/2,j}$ = 2.5D-SMMM$(A_{i+\sqrt{p}/2,j}, A_{ij}, A_{i+\sqrt{p}/2,j}, \Pi_{211}, n/2, p/8, c/2)$

// Update paths to V_2 via paths from V_2 to V_1 and back to V_2

10 $\Pi_{121}[i, j]$ sends $A_{i,j+\sqrt{p}/2}$ to $\Pi_{221}[i, j]$.

11 $\Pi_{211}[i, j]$ sends $A_{i+\sqrt{p}/2,j}$ to $\Pi_{221}[i, j]$.

12 $A_{i+\sqrt{p}/2,j+\sqrt{p}/2}$ = 2.5D-SMMM$(A_{i+\sqrt{p}/2,j}, A_{i,j+\sqrt{p}/2}, A_{i+\sqrt{p}/2,j+\sqrt{p}/2}, \Pi_{221}, n/2, p/8, c/2)$

// Find all-pairs shortest paths between vertices in V_2

13 $A_{i+\sqrt{p}/2,j+\sqrt{p}/2}$ = 2.5D-BLOCKED-DC-APSP$(A_{i+\sqrt{p}/2,j+\sqrt{p}/2}, \Pi_{221}, n/2, p/8, c/2)$

// Find shortest paths paths from V_2 to V_1

14 $\Pi_{221}[i, j]$ sends $A_{i+\sqrt{p}/2,j+\sqrt{p}/2}$ to $\Pi_{211}[i, j]$.

15 $A_{i+\sqrt{p}/2,j}$ = 2.5D-SMMM$(A_{i+\sqrt{p}/2,j+\sqrt{p}/2}, A_{i+\sqrt{p}/2,j}, A_{i+\sqrt{p}/2,j}, \Pi_{211}, n/2, p/8, c/2)$

// Find all-pairs shortest paths for vertices in V_1

16 $\Pi_{121}[i, j]$ sends $A_{i,j+\sqrt{p}/2}$ to $\Pi_{111}[i, j]$.

17 $\Pi_{211}[i, j]$ sends $A_{i+\sqrt{p}/2,j}$ to $\Pi_{111}[i, j]$.

18 A_{ij} = 2.5D-SMMM$(A_{i,j+\sqrt{p}/2}, A_{i+\sqrt{p}/2,j}, A_{ij}, \Pi_{111}, n/2, p/8, c/2)$

Fig. 11. A blocked parallel divide-and-conquer algorithm for the all-pairs shortest-paths problem